The C++ Scalar, Vector, Matrix and Tensor classes

E. Robert Tisdale

January 15, 2002

Abstract

This document describes an application programmer's interface standard for class libraries that provide high performance arithmetic on dense vector, matrix and tensor objects to support portable scientific and engineering applications including digital signal and image processing applications. The interface is easy to learn and use because the design follows simple, regular rules with a few exceptions that application programmers can quickly memorize. It provides a small number of simple functions which are combined in expressions to compute more complicated functions and simply extends many familiar C language scalar operations and functions to vector, matrix and tensor objects element by element. The C++ Scalar, Vector, Matrix and Tensor classes permit programmers to write reliable high-performance applications clearly and concisely so that they are easy to read, understand and maintain. They are designed to detected most programming errors at compile-time. Optional run-time error checking code can be compiled and/or linked into the application program to detect any remaining programming errors during testing.

Only Scalar and Vector classes support an offset type but all Scalar, Vector, Matrix and Tensor classes support a boolean type. Real Scalar, Vector, Matrix and Tensor classes are specified for both signed and unsigned built-in integral types `char`, `short int`, `int` and `long int`. Both real and complex Scalar, Vector, Matrix and Tensor classes are specified for built in floating-point types `float`, `double` and `long double`. Some non-standard implementations may also provide both real and complex fixed-point Scalar, Vector, Matrix and Tensor classes.

Contents

1 Introduction

It isn't practical for any single organization to develop and maintain a high performance scalar, vector, matrix and tensor arithmetic library that will port to every computing platform which is targeted for scientific and engineering applications. The C++ SVMT (Scalar, Vector, Matrix and Tensor) class library standard allows development and maintenance costs to be distributed among a variety of independent SVMT class library developers who provide application programmers with a portable application development environment which may include an editor, compiler, debugger, profiler, manual and other tools along with high-performance compile-time and run-time libraries for a particular platform or a restricted class of computer architectures.

The purpose of the SVMT class library standard is to support portable application programs. Most SVMT class library implementations for general purpose computing platforms are expected to provide all of the functionality specified by the standard. However, implementations intended for application development on special purpose computing platforms such as embedded processors may impose additional restrictions on application programs. For example, an SVMT class library for an embedded processor might restrict application programs to floating-point type `float` and provide no support for floating-point types `double` or `long double`. An application programmer can still export all application programs developed on the special purpose computer but some additional programming may be required to import application programs developed on other computers.

The C++ SVMT class library standard supports the first four tensor orders: 0. scalar, 1. vector, 2. matrix and 3. tensor. Some non-standard implementations may support higher order tensors. Both real and complex tensors are stored in contiguous one dimensional arrays of real numbers. The actual representation of these arrays might be hidden from the application programmer and it may not be possible to reference them through a pointer because, for example, the array may be distributed across several nodes in a multi-processor system. Complex numbers are stored as pairs of real numbers with the imaginary part following immediately after the real part. The real and imaginary parts of a complex tensor correspond to the even and odd elements of the real one dimensional array respectively. A scalar is a single real or complex element of a vector with n columns, a matrix with m rows and n columns or a tensor with l pages, m rows and n columns. Matrices are stored in row major order and tensors are stored in page major order.

SVMT classes are designed to detect most common programming errors at compile-time but some programming errors cannot be detected until run-time. The SVMT class library standard does not require any run-time error detection. Application programmers are responsible for all run-time error detection and handling. An SVMT class library developer may provide the application programmer with tools to help detect and debug programming errors at run-time but it probably won't help much to detect them after the application is placed into service.

The C++ SVMT class names <Type><System><Name> have three fields. The third field <Name> designates one of five base classes or one of the seven derived classes from the following table:

order	base	derived	
	Handle		
0	SubScalar	SubArray0	
1	SubVector	SubArray1	Vector
2	SubMatrix	SubArray2	Matrix
3	SubTensor	SubArray3	Tensor

The second field <System> is replaced by Complex for complex numbers or by the empty string for real numbers.

real	complex
<type>	<Type>Complex
<Type><Name>	<Type>Complex<Name>

The first field <Type> is a string from the following table:

<type>	<Type>	Complex	Handle	SubScalar	SubVector	SubMatrix	SubTensor
Offset	offset		*	*	*		
bool	bool		*	*	*	*	*
signed char	signed_char		*	*	*	*	*
unsigned char	unsigned_char		*	*	*	*	*
signed short int	signed_short_int		*	*	*	*	*
unsigned short int	unsigned_short_int		*	*	*	*	*
signed int	signed_int		*	*	*	*	*
unsigned int	unsigned_int		*	*	*	*	*
signed long int	signed_long_int		*	*	*	*	*
unsigned long int	unsigned_long_int		*	*	*	*	*
float	float	*	*	*	*	*	*
double	double	*	*	*	*	*	*
long double	long_double	*	*	*	*	*	*

which corresponds to one of the built-in types <type>. An asterisk * indicates which types are supported by each class. An SVMT class library implementation defines three synonyms for integral types:

unsigned int	Offset
unsigned int	Extent
signed int	Stride

An `Offset` or `Extent` may be an unsigned integral type but a `Stride` must be a signed integral type.

The SVMT class library standard specifies `class <Type>Complex` for each of the corresponding built-in floating-point types. The real and imaginary parts are represented by an array of two real numbers and the application programmer can always obtain a pointer to this array. If z is a complex number, then `&z.real()` is a pointer to the real part and `&z.real()+1` is a pointer to the imaginary part.

`Handle` objects contain a reference to a one dimensional array of type `<type>`. Handles are normally created and initialized by other SVMT classes but the application programmer can create a copy `<Type><System>Handle g = h` of an existing handle `h`. The application programmer can retrieve the element at offset j from the beginning of the array using member function `get(j)` or assign the value x to the element at offset j from the beginning of the array using member function `put(j, x)`.

`SubScalar`, `SubVector`, `SubMatrix` and `SubTensor` objects contain a handle, an offset from the beginning of the one dimensional array referenced by the handle to the first element of the subtensor and an extent and stride for each dimension. These data members are private and may be retrieved (but not modified) by the application programmer through the following member functions:

class				return type
`SubScalar`	`SubVector`	`SubMatrix`	`SubTensor`	`const`
`handle()`	`handle()`	`handle()`	`handle()`	`Handle&`
`offset()`	`offset()`	`offset()`	`offset()`	`Offset`
	`extent()`	`extent1()`	`extent1()`	`Extent`
	`stride()`	`stride1()`	`stride1()`	`Stride`
		`extent2()`	`extent2()`	`Extent`
		`stride2()`	`stride2()`	`Stride`
			`extent3()`	`Extent`
			`stride3()`	`Stride`

These simple attributes permit application programmers to reference slices which are a restricted but important set of subtensors within any tensor.

The `SubArray0`, `SubArray1`, `SubArray2` and `SubArray3` classes are derived from `SubScalar`, `SubVector`, `SubMatrix` and `SubTensor` classes respectively in order to permit the application programmer to reference an ordinary one dimensional array with a subtensor. The `SubArray0` class exists so that a `SubArray1` subscript operator can return a `SubArray0` and appear on the left hand side of an assignment operation. The `SubScalar` class exists so that a `SubVector` subscript operator can return a `SubScalar` and appear on the left hand side of an assignment operation. The `Vector`, `Matrix` and `Tensor` classes are derived from `SubVector`, `SubMatrix` and `SubTensor` classes respectively but have constructors which automatically allocate a one dimensional array for the tensor object from free storage when it is created and destructors which automatically deallocate the one dimensional array when it is destroyed. Most functions and operations are defined on `SubVector`, `SubMatrix` and `SubTensor` classes and inherited by the corresponding derived classes.

2 Class Libraries

2.1 Complex classes

The SVMT Complex class libraries are based upon math libraries for built-in floating-point types `float`, `double` and `long double` which include support for inverse trigonometric and hyperbolic functions.

2.1.1 Constructors

Default Constructors

`<Type>Complex c`

create an uninitialized complex number `c`.

Explicit Constructors

`<Type>Complex c = r` and
`<Type>Complex c(r, i)`

create a complex number `c` and copy the real and imaginary parts from real values `r` and `i` respectively. If `i` is omitted, the imaginary part of `c` is set to zero.

Copy Constructors

`<Type>Complex z = c`

create a new complex number `z` and copy the respective real and imaginary parts from complex number `c`.

2.1.2 Functions

There are only twenty three complex functions:

`z.real()`	$\Re(z)$	`z.imag()`	$\Im(z)$		
`conj(z)`	z^*	`iconj(z)`	iz^*		
`norm(z)`	$\|z\|^2$	`sqrt(z)`	$\sqrt{z}$		
`exp(z)`	e^z	`log(z)`	$\ln(z)$		
`abs(z)`	$\|z\|$	`arg(z)`	$\arg(z)$	`polar(r, t)`	re^{it}
`cos(z)`	$\cos(z)$	`sin(z)`	$\sin(z)$	`tan(z)`	$\tan(z)$
`acos(z)`	$\cos^{-1}(z)$	`asin(z)`	$\sin^{-1}(z)$	`atan(z)`	$\tan^{-1}(z)$
`cosh(z)`	$\cosh(z)$	`sinh(z)`	$\sinh(z)$	`tanh(z)`	$\tanh(z)$
`acosh(z)`	$\cosh^{-1}(z)$	`asinh(z)`	$\sinh^{-1}(z)$	`atanh(z)`	$\tanh^{-1}(z)$

Member functions z.real() and z.imag() return a reference to the real and imaginary parts of z respectively so they may appear on the left hand side of an assignment operation. Functions abs(z), arg(z) and norm(z) return a real number of type <type>. All of the other functions return a complex number of type <Type>Complex. Because there is no imaginary number class, function iconj(z) is included to help compensate.

2.1.3 Operators

Unary operators

	minus		plus	
unary	-z	$-z$	+z	$+z$

return type <Type>Complex.

Binary operators

	real–complex		complex–complex		complex–real	
multiply	r*z	$r \cdot z$	z*c	$z \cdot c$	z*r	$z \cdot r$
divide	r/z	r/z	z/c	z/c	z/r	z/r
add	r + z	$r + z$	z + c	$z + c$	z + r	$z + r$
subtract	r - z	$r - z$	z - c	$z - c$	z - r	$z - r$

return type <Type>Complex.

Output operator

return type	function	functionality
ostream&	cout << z	cout << '(' << z.real() << ',' << ' ' << z.imag() << ')'

Input operator

return type	function	functionality
istream&	cin >> z	cin >> '(' >> z.real() >> ',' >> z.imag() >> ')'

Comparison operators

	real–complex		complex–complex		complex–real	
equal	r == z	$r = z$	z == c	$z = c$	z == r	$z = r$
not equal	r != z	$r \neq z$	z != c	$z \neq c$	z != r	$z \neq r$

return type bool.

Assignment operators

	complex–complex		complex–real	
simple	`z  = c`	$z \leftarrow c$	`z  = r`	$z \leftarrow r$
multiply	`z *= c`	$z \leftarrow z \cdot c$	`z *= r`	$z \leftarrow z \cdot r$
divide	`z /= c`	$z \leftarrow z/c$	`z /= r`	$z \leftarrow z/r$
add	`z += c`	$z \leftarrow z + c$	`z += r`	$z \leftarrow z + r$
subtract	`z -= c`	$z \leftarrow z - c$	`z -= r`	$z \leftarrow z - r$

return a reference of type `<Type>Complex&` to the left hand side.

2.2 Handle classes

A handle might be represented by a pointer to an ordinary one dimensional array of real numbers but more generally it will be a reference to some implementation dependent data structure which is hidden from the application programmer.

2.2.1 Constructors

Private Constructors

```
<Type><System>Handle h(p)
```

where `p` is a pointer to an ordinary one dimensional array of type `<type>`, are used by the SVMT class library developer to create a handle.

Public Copy Constructors

```
<Type><System>Handle g(h)
```

make a copy of the handle but not the underlying one dimensional array of real numbers.

2.2.2 Functions

Member function `h.empty()` returns true if the handle contains a null reference to the one dimensional array. Member function `h.get(j)` returns element `j`. Member function `h.put(j, x)` assigns the value of scalar `x` to element `j`. If `h` is complex and `x` is real, then the imaginary part of the element at offset `j` is set to zero. Element `j` is located at an offset of j from the beginning of the one dimensional array if `h` is real or at an offset of $2j$ from the beginning of the one dimensional array if `h` is complex.

2.2.3 Operators

The cast (type conversion) operator (<type>*)h returns 0 unless the handle h contains a pointer to an ordinary one dimensional array of real numbers. If h is complex, then the cast (type conversion) operator (<Type>Handle)h returns a real handle on the same one dimensional array referenced through handle h.

2.3 SubScalar, SubVector, SubMatrix and SubTensor classes

SubScalar, SubVector, SubMatrix and SubTensor objects contain a handle and an offset from the first element of the real one dimensional array referenced though the handle to the first element of the SubScalar, SubVector, SubMatrix or SubTensor. SubVector, SubMatrix and SubTensor objects also contain the number of elements and the stride between elements for each dimension. These simple attributes permit application programmers to access a restricted but very important set of subtensors without sacrificing performance.

2.3.1 Constructors

Explicit Constructors

<Type><System>SubScalar s(h, o),
<Type><System>SubVector v(h, o, n1, s1),
<Type><System>SubMatrix M(h, o, n2, s2, n1, s1) and
<Type><System>SubTensor T(h, o, n3, s3, n2, s2, n1, s1)

where h is a handle of type <Type><System>Handle, o is an offset of type Offset, n1, n2 and n3 are extents of type Extent and s1, s2 and s3 are strides of type Stride. SVMT class libraries do not normally check whether the one dimensional array referenced through h actually contains the subtensor or not.

Copy Constructors

<Type><System>SubScalar t(s),
<Type><System>SubVector w(v),
<Type><System>SubMatrix N(M) and
<Type><System>SubTensor U(T)

simply copy the attributes of an existing subtensor not the underlying one dimensional array.

Default Constructors

<Type><System>SubVector v,
<Type><System>SubMatrix M and
<Type><System>SubTensor T

create an empty subvector, submatrix and subtensor respectively.

2.3.2 Functions

Static data members: All `SubVector` classes include static member functions

return type	function
`Extent&`	<Type><System>SubVector::columns()

which reference the number of columns displayed on each line using `operator <<`. The I/O stream manipulator function `setw(w, n)` has been overloaded to accept a second integer argument `n` which specifies the number of columns displayed on each line for the following SubVector, SubMatrix or SubTensor object and should be used instead of static member function `columns()`.

Data members: All `SubScalar`, `SubVector`, `SubMatrix` and `SubTensor` classes include member functions

class				return type
`SubScalar`	`SubVector`	`SubMatrix`	`SubTensor`	`const`
`s.handle()`	`v.handle()`	`M.handle()`	`T.handle()`	`Handle&`
`s.offset()`	`v.offset()`	`M.offset()`	`T.offset()`	`Offset`
	`v.extent()`	`M.extent1()`	`T.extent1()`	`Extent`
	`v.stride()`	`M.stride1()`	`T.stride1()`	`Stride`
		`M.extent2()`	`T.extent2()`	`Extent`
		`M.stride2()`	`T.stride2()`	`Stride`
			`T.extent3()`	`Extent`
			`T.stride3()`	`Stride`

which return a constant reference to the handle, the offset from the first element of the one dimensional array to the first element of the subtensor and the extent and stride for each dimension.

Empty: All `SubScalar`, `SubVector`, `SubMatrix` and `SubTensor` classes include constant member functions

return type	function	functionality
`bool`	`s.empty()`	`s.handle().empty()`,
`bool`	`v.empty()`	`v.handle().empty()` $\vee\ n_1 = 0$,
`bool`	`M.empty()`	`M.handle().empty()` $\vee\ n_1 = 0\ \vee\ n_2 = 0$ and
`bool`	`T.empty()`	`T.handle().empty()` $\vee\ n_1 = 0\ \vee\ n_2 = 0\ \vee\ n_3 = 0$

respectively which return true if the subtensor references a null one dimensional array or if any of the extents are zero.

Allocate: All SubVector, SubMatrix and SubTensor classes include static member functions

return type	allocate memory
<Type><System>Handle	allocate(n),
<Type><System>Handle	allocate(m, n) and
<Type><System>Handle	allocate(l, m, n)

respectively which allocate memory for a one dimensional array of length n, m*n and l*m*n respectively. Application programmers should avoid these functions if possible.

Free: All SubVector, SubMatrix and SubTensor classes include member functions

return type	free memory
<Type><System>SubVector&	v.free(),
<Type><System>SubMatrix&	M.free() and
<Type><System>SubTensor&	T.free()

respectively which de-allocate the memory allocated by allocate(n), allocate(m, n) and allocate(l, m, n) respectively. Application programmers should avoid these functions if possible.

Resize: All SubVector, SubMatrix and SubTensor classes include member functions

return type	explicit resize
<Type><System>SubVector&	v.resize(h, o, n1, s1),
<Type><System>SubMatrix&	M.resize(h, o, n2, s2, n1, s1) and
<Type><System>SubTensor&	T.resize(h, o, n3, s3, n2, s2, n1, s1)

respectively which correspond to the respective explicit constructors. Application programmers should avoid these functions if possible.

All SubVector, SubMatrix and SubTensor classes include member functions

return type	copy resize
<Type><System>SubVector&	w.resize(v),
<Type><System>SubMatrix&	N.resize(M) and
<Type><System>SubTensor&	U.resize(T)

respectively which correspond to the respective copy constructors. Application programmers should avoid these functions if possible.

All SubVector, SubMatrix and SubTensor classes include member functions

return type	default resize
<Type><System>SubVector&	`v.resize()`,
<Type><System>SubMatrix&	`M.resize()` and
<Type><System>SubTensor&	`T.resize()`

respectively which correspond to the respective default constructors. Application programmers should avoid these functions if possible.

Subtensors of subtensors: All `SubVector` classes except `offsetSubVector` and all `SubMatrix` and `SubTensor` classes include member functions

return type	subsubtensor
<Type><System>SubVector	`v.sub(j, n1, s1)`,
<Type><System>SubMatrix	`M.sub(i, n2, s2)`,
<Type><System>SubTensor	`T.sub(h, n3, s3)`,
<Type><System>SubMatrix	`M.sub(i, n2, s2, j, n1, s1)`,
<Type><System>SubTensor	`T.sub(h, n3, s3, i, n2, s2)` and
<Type><System>SubTensor	`T.sub(h, n3, s3, i, n2, s2, j, n1, s1)`

respectively which return a subtensor to reference a subtensor of a subtensor computing the attributes as shown in the following tables:

`v.sub(j, n1, s1)`	
`handle()`	= `v.handle()`
`offset()`	= `v.offset()+j*v.stride()`
`extent()`	= `n1`
`stride()`	= `s1*v.stride()`

`M.sub(i, n2, s2)`	
`handle()`	= `M.handle()`
`offset()`	= `M.offset()+i*M.stride2()`
`extent1()`	= `M.extent1()`
`stride1()`	= `M.stride1()`
`extent2()`	= `n2`
`stride2()`	= `s2*M.stride2()`

`T.sub(h, n3, s3)`	
`handle()`	= `T.handle()`
`offset()`	= `T.offset()+h*T.stride3()`
`extent1()`	= `T.extent1()`
`stride1()`	= `T.stride1()`
`extent2()`	= `T.extent2()`
`stride2()`	= `T.stride2()`
`extent3()`	= `n3`
`stride3()`	= `s3*T.stride3()`

M.sub(i, n2, s2, j, n1, s1)	
handle()	= M.handle()
offset()	= M.offset()+i*M.stride2()+j*M.stride1()
extent1()	= n1
stride1()	= s1*M.stride1()
extent2()	= n2
stride2()	= s2*M.stride2()

T.sub(h, n3, s3, i, n2, s2)	
handle()	= T.handle()
offset()	= T.offset()+h*T.stride3()+i*T.stride2()
extent1()	= T.extent1()
stride1()	= T.stride1()
extent2()	= n2
stride2()	= s2*T.stride2()
extent3()	= n3
stride3()	= s3*T.stride3()

T.sub(h, n3, s3, i, n2, s2, j, n1, s1)	
handle()	= T.handle()
offset()	= T.offset()+h*T.stride3()+i*T.stride2()+j*T.stride1()
extent1()	= n1
stride1()	= s1*T.stride1()
extent2()	= n2
stride2()	= s2*T.stride2()
extent3()	= n3
stride3()	= s3*T.stride3()

where `h`, `i` and `j` are of type `Offset`, `n1`, `n2` and `n3` are of type `Extent` and `s1`, `s2` and `s3` are of type `Stride`. These functions do not normally check whether the subtensor actually contains the subsubtensor or not.

Containment: All `SubVector` classes except `offsetSubVector` and all `SubMatrix` and `SubTensor` classes include constant member functions

bool	v.contains(j, n1, s1),
bool	M.contains(i, n2, s2),
bool	T.contains(h, n3, s3),
bool	M.contains(i, n2, s2, j, n1, s1),
bool	T.contains(h, n3, s3, i, n2, s2) and
bool	T.contains(h, n3, s3, i, n2, s2, j, n1, s1)

which return true if

`v` contains `v.sub(j, n1, s1)`,
`M` contains `M.sub(i, n2, s2)`,
`T` contains `T.sub(h, n3, s3)`,

`M` contains `M.sub(i, n2, s2, j, n1, s1)`,
`T` contains `T.sub(h, n3, s3, i, n2, s2)` and

`T` contains `T.sub(h, n3, s3, i, n2, s2, j, n1, s1)`

respectively which means that

$$\begin{array}{l}
0 \le j < \texttt{extent()} \;\wedge\; 0 \le j + (n_1 - 1)s_1 < \texttt{extent()}, \\
0 \le j < \texttt{extent1()} \;\wedge\; 0 \le j + (n_1 - 1)s_1 < \texttt{extent1()}, \\
0 \le i < \texttt{extent2()} \;\wedge\; 0 \le i + (n_2 - 1)s_2 < \texttt{extent2()} \text{ and} \\
0 \le h < \texttt{extent3()} \;\wedge\; 0 \le h + (n_3 - 1)s_3 < \texttt{extent3()}.
\end{array}$$

Order promotion: All `SubScalar` classes except `offsetSubScalar` include constant member functions

return type	function
const <Type><System>SubVector	`s.subvector(n)`,
const <Type><System>SubMatrix	`s.submatrix(n, m)` and
const <Type><System>SubTensor	`s.subtensor(n, m, l)`

which return subvectors with n columns, submatrices with n columns and m rows or subtensors with n columns, m rows and l pages but `stride3()` = `stride2()` = `stride1()` = `stride()` = 0 so that every element references the same subscalar. Arguments `n`, `m` and `l` are of type `Extent` and default to a value of +1 if omitted.

All `SubVector` classes except `offsetSubVector` include constant member functions

return type	function
const <Type><System>SubMatrix	`v.submatrix(m)` and
const <Type><System>SubTensor	`v.subtensor(m, l)`

which return submatrices with m rows or subtensors with m rows and l pages but `stride3()` = `stride2()` = 0 so that every row references the same subvector. Arguments `m` and `l` are of type `Extent` and default to a value of +1 if omitted.

All `SubMatrix` classes include constant member functions

return type	function
const <Type><System>SubTensor	`M.subtensor(l)`

which return subtensors with l pages but `stride3()` = 0 so that every page references the same submatrix. Argument `l` is of type `Extent` and defaults to a value of +1 if omitted.

Transpose: All `SubMatrix` classes include member functions

result	return type	function
page matrices	`<Type><System>SubMatrix`	`M.t()`

which return a submatrix to reference the transpose of the original submatrix.

All `SubTensor` classes include member functions

result	return type	function
page matrices	`<Type><System>SubTensor`	`T.t12()`,
column matrices	`<Type><System>SubTensor`	`T.t23()` and
row matrices	`<Type><System>SubTensor`	`T.t31()`

which return a subtensor to reference the transpose of page, column and row matrices of the original subtensor respectively.

Diagonal: All `SubMatrix` classes include member functions

result	return type	function
page matrices	`<Type><System>SubVector`	`M.diag()`

which return a subvector to reference the diagonal of the original submatrix.

All `SubTensor` classes include member functions

result	return type	function
page matrices	`<Type><System>SubMatrix`	`T.diag12()`,
column matrices	`<Type><System>SubMatrix`	`T.diag23()` and
row matrices	`<Type><System>SubMatrix`	`T.diag31()`

which return a submatrix to reference the diagonals of page, column and row matrices of the original subtensor respectively.

Reverse: All `SubVector` classes except `offsetSubVector` include member functions

return type	function	o	s_1
`<Type><System>SubVector`	`v.r()`	$o + s_1(n_1 - 1)$	$-s_1$

which return a subvector to reference the columns of the original subvector in reverse order.

Member function `v.reverse()` actually reverses the elements in place and returns a reference to the original subvector.

All `SubMatrix` classes include member functions

return type	function	o	s_1	s_2
`<Type><System>SubMatrix`	`M.r1()`	$o + s_1(n_1 - 1)$	$-s_1$	$+s_2$
`<Type><System>SubMatrix`	`M.r2()`	$o + s_2(n_2 - 1)$	$+s_1$	$-s_2$
`<Type><System>SubMatrix`	`M.r()`	$o + s_1(n_1 - 1) + s_2(n_2 - 1)$	$-s_1$	$-s_2$

which return a submatrix to reference the columns or rows or both the columns and rows of the original submatrix in reverse order.

Member functions M.reverse1(), M.reverse2() and M.reverse() actually reverse the elements in place and return a reference to the original submatrix.

All SubTensor classes include member functions

return type	function	o	s_1	s_2	s_3
<Type><System>SubTensor	T.r1()	$o + s_1(n_1 - 1)$	$-s_1$	$+s_2$	$+s_3$
<Type><System>SubTensor	T.r2()	$o + s_2(n_2 - 1)$	$+s_1$	$-s_2$	$+s_3$
<Type><System>SubTensor	T.r3()	$o + s_3(n_3 - 1)$	$+s_1$	$+s_2$	$-s_3$
<Type><System>SubTensor	T.r()	$o + s_1(n_1 - 1) + s_2(n_2 - 1) + s_3(n_3 - 1)$	$-s_1$	$-s_2$	$-s_3$

which return a subtensor to reference the columns, rows or pages or the columns, rows and pages of the original subtensor in reverse order.

Member functions T.reverse1(), T.reverse2(), T.reverse3() and T.reverse() actually reverse the elements in place and return a reference to the original subtensor.

Even and odd columns: All SubVector classes except offsetSubVector and all SubMatrix and SubTensor classes include member functions

return type	even	odd
<Type><System>SubVector	v.even()	v.odd(),
<Type><System>SubMatrix	M.even()	M.odd() and
<Type><System>SubTensor	T.even()	T.odd()

respectively which return a subtensor to reference the even or odd elements in each row of the original subtensor.

Real and imaginary parts: All complex floating-point SubVector, SubMatrix and SubTensor classes include member functions

return type	real	imaginary
<Type>SubVector	v.real()	v.imag(),
<Type>SubMatrix	M.real()	M.imag() and
<Type>SubTensor	T.real()	T.imag()

respectively which return a real subtensor to reference the real or imaginary part.

Discrete Fourier transform: All complex floating-point SubVector, SubMatrix and SubTensor classes include member functions

return type	complex to complex
<Type>ComplexSubVector&	v.dft(sign),
<Type>ComplexSubMatrix&	M.dft(sign) and
<Type>ComplexSubTensor&	T.dft(sign)

respectively which perform a complex to complex Discrete Fourier Transform of the row vectors in-place and return a reference to the transformed subtensor. Argument `sign` is of type `signed int`, defaults to a value of -1 if omitted and determines whether

$$v_k \leftarrow \begin{cases} \sum_{j=0}^{n-1} v_j e^{-i2\pi jk/n} & \text{if } \texttt{sign} < 0 \\ \sum_{j=0}^{n-1} v_j e^{+i2\pi jk/n} & \text{if } 0 \leq \texttt{sign} \end{cases} \tag{1}$$

is a direct or inverse Discrete Fourier Transform of row vector v.

All real floating-point `SubVector`, `SubMatrix` and `SubTensor` classes include member functions

return type	complex to real	real to complex
<Type>SubVector&	v.rdft(sign)	v.cdft(sign),
<Type>SubMatrix&	M.rdft(sign)	M.cdft(sign) and
<Type>SubTensor&	T.rdft(sign)	T.cdft(sign)

respectively which perform a complex to real or real to complex Discrete Fourier Transform of the row vectors in-place and return a reference to the transformed subtensor. The first half of complex row vector v is packed into real row vector v with $\Im(v_0) = \Re(v_{n/2})$ if the extent n is even or $\Im(v_0) = \Im(v_{(n-1)/2})$ if the extent n is odd.

All real and complex `SubVector` classes include a member function

return type	function	functionality
<Type>SubVector&	u.pack(v)	complex into real and
<Type>ComplexSubVector&	v.pack(u)	real into complex

respectively which pack a complex vector v into a real vector u and pack a real vector u into a complex vector v respectively so that $u_{2j} = \Re\{v_j\}\forall j|0 \leq 2j < n$ and $u_{2j+1} = \Im\{v_j\}\forall j|1 \leq 2j+1 < n$ where n is the length of real vector u which must be no more than twice the length of complex vector v.

Ramp: All real `SubVector` classes except `boolSubVector` and all `SubMatrix` and `SubTensor` classes include member functions

return type	function	functionality
<Type>SubVector&	v.ramp()	$v_j \leftarrow j$,
<Type>SubMatrix&	M.ramp()	$M_{i,j} \leftarrow j$ and
<Type>SubMatrix&	T.ramp()	$T_{h,i,j} \leftarrow j$

respectively which initialize each element to the corresponding column index then return a reference to the subtensor.

Swap: All `SubVector`, `SubMatrix` and `SubTensor` classes include member functions

return type	function	functionality
`<Type><System>SubVector&`	`v.swap(j, k)`	$v_j \longleftrightarrow v_k$,
`<Type><System>SubMatrix&`	`M.swap(i, k)`	$M_{i,j} \longleftrightarrow M_{k,j}$ and
`<Type><System>SubTensor&`	`T.swap(h, k)`	$T_{h,i,j} \longleftrightarrow T_{k,i,j}$

respectively which swap column j with column k, row i with row k and page h with page k respectively then return a reference to the subvector, submatrix and subtensor respectively.

All `SubVector` classes except `offsetSubVector` and all `SubMatrix` and `SubTensor` classes include member functions

return type	function	functionality
`<Type><System>SubVector&`	`v.swap(w)`	$v_j \longleftrightarrow w_j$,
`<Type><System>SubMatrix&`	`M.swap(N)`	$M_{i,j} \longleftrightarrow N_{i,j}$ and
`<Type><System>SubTensor&`	`T.swap(U)`	$T_{h,i,j} \longleftrightarrow U_{h,i,j}$

respectively which swap the elements of two subtensors then return a reference to the first subtensor. Both operands must be the same size.

Rotate and shift: All `SubVector`, `SubMatrix` and `SubTensor` classes except `offsetSubVector` include member functions

return type	function	functionality
`<Type><System>SubVector&`	`v.rotate(n)`	$v_{(j+n) \bmod \texttt{v.extent()}}$,
`<Type><System>SubMatrix&`	`M.rotate(n)`	$M_{i,(j+n) \bmod \texttt{M.extent1()}}$ and
`<Type><System>SubTensor&`	`T.rotate(n)`	$T_{h,i,(j+n) \bmod \texttt{T.extent1()}}$

respectively which execute an in-place circular shift on all row vectors then return a reference to the rotated subtensor.

All `SubVector`, `SubMatrix` and `SubTensor` classes except `offsetSubVector` include member functions

return type	function	functionality
`<Type><System>SubVector&`	`v.shift(n, s)`	v_{j+n} $\;0 \le j+n <$ `v.extent()` s otherwise,
`<Type><System>SubMatrix&`	`M.shift(n, s)`	$M_{i,j+n}$ $\;0 \le j+n <$ `M.extent1()` s otherwise and
`<Type><System>SubTensor&`	`T.shift(n, s)`	$T_{h,i,j+n}$ $\;0 \le j+n <$ `T.extent1()` s otherwise

respectively which execute an in-place shift on all row vectors then return a reference to the shifted subtensor. Scalar argument `s` defaults to a value of 0 if omitted.

Minimum and maximum: All real SubVector classes except boolSubVector include constant member functions

return type	minimum	maximum
Offset	v.min()	v.max()

which return an index to the minimum or maximum element and functions

return type	minimum	maximum
<type>	min(v)	max(v)

which return the minimum or maximum element itself.

All real SubMatrix and SubTensor class libraries include functions

return type	minimum	maximum
const <Type>Vector	min(M)	max(M) and
const <Type>Matrix	min(T)	max(T)

respectively which return the minimum or maximum element in each row. In order to find the minimum of all of the elements in a submatrix, the application programmer writes min(min(M)) and min(min(min(T))) in order to find the minimum of all of the elements in a subtensor. In order to find the maximum of all of the elements in a submatrix, the application programmer writes max(max(M)) and max(max(max(T))) in order to find the maximum of all of the elements in a subtensor.

All real SubVector, SubMatrix and SubTensor class libraries except offsetSubVector and boolSubVector include functions

return type	minimum	maximum
const <Type>Vector	min(v, w)	max(v, w),
const <Type>Matrix	min(M, N)	max(M, N) and
const <Type>Tensor	min(T, U)	max(T, U)

respectively which compare corresponding elements of a pair of subvectors, submatrices or subtensors and return the minimum or maximum. Both arguments must be the same size.

Row vector sum: All SubVector classes except offsetSubVector and boolSubVector include constant member functions

	return type			
result	real	complex	function	functionality
scalar	<type>	<Type>Complex	v.sum()	$\sum_{j=0}^{n-1} v_j$ n=v.extent().

All SubMatrix and SubTensor classes except boolSubMatrix and boolSubTensor include constant member functions

return type	function	functionality
const <Type><System>Vector	M.sum()	$\sum_{j=0}^{n-1} M_{i,j}$ $n=$M.extent1() and
const <Type><System>Matrix	T.sum()	$\sum_{j=0}^{n-1} T_{h,i,j}$ $n=$T.extent1()

respectively. In order to sum all of the elements in a submatrix, the application programmer writes M.sum().sum() and T.sum().sum().sum() in order to sum all of the elements in a subtensor.

[Transpose] dot product: All SubVector classes except offsetSubVector and boolSubVector include constant member functions

	return type		
result	real	complex	functionality
scalar	<type>	<Type>Complex	v.dot(w) $= vw^T$.

All SubVector classes except offsetSubVector and boolSubVector include constant member functions

result	return type	functionality
row vector	const <Type><System>Vector	v.dot(M) $= vM^T$.

All SubMatrix classes except offsetSubMatrix and boolSubMatrix include constant member functions

result	return type	functionality
column vector	const <Type><System>Matrix	M.dot(v) $= Mv^T$ and
matrix	const <Type><System>Matrix	M.dot(N) $= MN^T$.

All SubVector classes except boolSubVector and offsetSubVector include constant member functions

	return type		
result	real	complex	functionality
scalar	<type>	<Type>Complex	v.dot() $= vv^T$.

All SubMatrix classes except boolSubMatrix include constant member functions

result	return type	functionality
square matrix	const <Type><System>Square	M.dot() $= MM^T$.

All SubMatrix classes include member functions

function	functionality
M.pack(s)	$M_{i,j} \leftarrow s \quad \forall j \leq i$ and
M.pack(N)	$M_{i,j} \leftarrow N_{i,j} \quad \forall j \leq i$

which pack the diagonal and subdiagonal elements of submatrix M with scalar s and the corresponding elements of submatrix N respectively then return a reference to submatrix M. The scalar $s = 0$ if omitted and the function is undefined unless submatrix M and submatrix N are exactly the same size.

Matrix decomposition: All real and complex floating-point `SubMatrix` classes include member functions

return type	function	solves
`const <Type><System>Matrix`	`M.svd()`	$M = UDV^T$ if $m \geq n$ $M^T = UDV^T$ if $m < n$

which decompose an $m \times n$ matrix M into orthonormal matrices U and V and a diagonal matrix D where $d_i = D_{i,i}$ are the singular values, replace M with U if $m \geq n$ or U^T if $m < n$ then return $\begin{bmatrix} d \\ V \end{bmatrix}$.

All real and complex floating-point `SubMatrix` classes include member functions

return type	function	solves	function	solves
`offsetVector`	`M.lud()`	$PM = L(DU)$	`M.qrd()`	$PM = QR$

which decompose matrix M in-place and return the permutation vector required to obtain the permutation PM of the rows of matrix M. Matrices L and Q are packed into the subdiagonal elements and matrices DU and R are packed in the remaining elements of matrix M. Orthogonal matrix Q is actually the product of Householder reflections and only the essential part of each normalized Householder vector is packed into the columns below the diagonal.

Triangular system solvers: All real and complex floating-point `SubVector` and `SubMatrix` classes include constant member functions

`const <Type><System>`	function	solves	function	solves
`Vector`	`v.pl(p, L)`	$v = w(P^TL)^T$	`v.l(L)`	$v = wL^T$ and
`Matrix`	`M.pl(p, L)`	$M = N(P^TL)^T$	`M.l(L)`	$M = NL^T$

respectively. The P^TL and L system solvers ignore the elements of L along and above the diagonal. They assume that all of the elements along the diagonal are ones and all of the elements above the diagonal are zero.

All real and complex floating-point `SubVector` and `SubMatrix` classes include constant member functions

`const <Type><System>`	function	solves	function	solves
`Vector`	`v.u(U)`	$v = wU^T$	`v.up(U, p)`	$v = w(UP)^T$ and
`Matrix`	`M.u(U)`	$M = NU^T$	`M.up(U, p)`	$M = N(UP)^T$

respectively. The U and UP system solvers ignore the elements of U along and below the diagonal. They assume that all of the elements along the diagonal are ones and all of the elements below the diagonal are zero.

All real and complex floating-point `SubVector` and `SubMatrix` classes include constant member functions

const <Type><System>	function	solves	function	solves
Vector	`v.pld(p, LD)`	$v = w(P^T LD)^T$	`v.ld(LD)`	$v = w(LD)^T$ and
Matrix	`M.pld(p, LD)`	$M = N(P^T LD)^T$	`M.ld(LD)`	$M = N(LD)^T$

respectively. The $P^T LD$ and LD system solvers ignore all the elements of LD above the diagonal and assume that they are zero.

All real and complex floating-point `SubVector` and `SubMatrix` classes include constant member functions

const <Type><System>	function	solves	function	solves
Vector	`v.du(DU)`	$v = w(DU)^T$	`v.dup(DU, p)`	$v = w(DUP)^T$ and
Matrix	`M.du(DU)`	$M = N(DU)^T$	`M.dup(DU, p)`	$M = N(DUP)^T$

respectively. The DU and DUP system solvers ignore all the elements of DU below the diagonal and assume that they are zero.

Triangular systems with m rows and n columns are under-determined if $m < n$, square if $m = n$ or over-determined if $m > n$. Only the first m permuted columns of the solution to an under-determined system are non-zero. The solution to an over-determined system depends only upon the first n permuted rows.

Orthogonal system solvers: All real and complex floating-point `SubVector` and `SubMatrix` classes include constant member functions

const <Type><System>	function	solves	function	solves
Vector	`v.pq(p, L)`	$v = w(P^T Q)^T$	`v.q(L)`	$v = wQ^T$ and
Matrix	`M.pq(p, L)`	$M = N(P^T Q)^T$	`M.q(L)`	$M = NQ^T$

respectively. The $P^T Q$ and Q orthogonal system solvers ignore the elements of L along and above the diagonal. They assume that all of the elements along the diagonal are ones and the essential part of each normalized Householder vector is stored in the columns of L below the diagonal.

Comparisons: All real `SubVector`, `SubMatrix` and `SubTensor` classes except `boolSubVector`, `boolSubMatrix`, `boolSubTensor` and `offsetSubVector` include constant member functions

const boolVector		const boolMatrix		const boolTensor	
`v.lt(s)`	$v_j < s$	`M.lt(s)`	$M_{i,j} < s$	`T.lt(s)`	$T_{h,i,j} < s$,
`v.le(s)`	$v_j \leq s$	`M.le(s)`	$M_{i,j} \leq s$	`T.le(s)`	$T_{h,i,j} \leq s$,
`v.gt(s)`	$v_j > s$	`M.gt(s)`	$M_{i,j} > s$	`T.gt(s)`	$T_{h,i,j} > s$ and
`v.ge(s)`	$v_j \geq s$	`M.ge(s)`	$M_{i,j} \geq s$	`T.ge(s)`	$T_{h,i,j} \geq s$

where s is a scalar and

`const boolVector`		`const boolMatrix`		`const boolTensor`	
`v.lt(w)`	$v_j < w_j$	`M.lt(N)`	$M_{i,j} < N_{i,j}$	`T.lt(U)`	$T_{h,i,j} < U_{h,i,j}$,
`v.le(w)`	$v_j \leq w_j$	`M.le(N)`	$M_{i,j} \leq N_{i,j}$	`T.le(U)`	$T_{h,i,j} \leq U_{h,i,j}$,
`v.gt(w)`	$v_j > w_j$	`M.gt(N)`	$M_{i,j} > N_{i,j}$	`T.gt(U)`	$T_{h,i,j} > U_{h,i,j}$ and
`v.ge(w)`	$v_j \geq w_j$	`M.ge(N)`	$M_{i,j} \geq N_{i,j}$	`T.ge(U)`	$T_{h,i,j} \geq U_{h,i,j}$

where both operands are the same size.

All `SubVector`, `SubMatrix` and `SubTensor` classes except `offsetSubVector` include constant member functions

`const boolVector`		`const boolMatrix`		`const boolTensor`	
`v.eq(s)`	$v_j = s$	`M.eq(s)`	$M_{i,j} = s$	`T.eq(s)`	$T_{h,i,j} = s$ and
`v.ne(s)`	$v_j \neq s$	`M.ne(s)`	$M_{i,j} \neq s$	`T.ne(s)`	$T_{h,i,j} \neq s$

where s is a scalar and

`const boolVector`		`const boolMatrix`		`const boolTensor`	
`v.eq(w)`	$v_j = w_j$	`M.eq(N)`	$M_{i,j} = N_{i,j}$	`T.eq(U)`	$T_{h,i,j} = U_{h,i,j}$ and
`v.ne(w)`	$v_j \neq w_j$	`M.ne(N)`	$M_{i,j} \neq N_{i,j}$	`T.ne(U)`	$T_{h,i,j} \neq U_{h,i,j}$

where both operands are the same size.

All `boolSubVector`, `boolSubMatrix` and `boolSubTensor` classes include functions

return type	OR		AND	
`bool`	`any(v)`	$\neg(v \quad = \text{false})$	`all(v)`	$v \quad = \text{true}$,
`boolVector`	`any(M)`	$\neg(M_i \quad = \text{false})$	`all(M)`	$M_i \quad = \text{true}$ and
`boolMatrix`	`any(T)`	$\neg(T_{h,i} \quad = \text{false})$	`all(T)`	$T_{h,i} \quad = \text{true}$

respectively.

Sparse subtensors: All `SubVector`, `SubMatrix` and `SubTensor` classes except `offsetSubVector` include constant member functions

return type	function	return type	function
`Extent`	`v.zeros()`	`const offsetVector`	`v.index()`,
`Extent`	`M.zeros()`	`const offsetVector`	`M.index()` and
`Extent`	`T.zeros()`	`const offsetVector`	`T.index()`

respectively which count the number of zero elements and find the non-zero elements in a subtensor. Function index searches the pages, rows and columns of a subtensor in order for the next non-zero element t_k and computes the index x_k

t_k	x_k
v_j	`j`
$M_{i,j}$	`j + i*M.extent1()`
$T_{h,i,j}$	`j + i*T.extent1() + h*T.extent2()*T.extent1()`

then returns the indices as a const offsetVector x.

All SubVector, SubMatrix and SubTensor classes except offsetSubVector include constant member functions

return type	function
const <Type><System>Vector	v.gather(x),
const <Type><System>Vector	M.gather(x) and
const <Type><System>Vector	T.gather(x)

respectively and member functions

return type	function
<Type><System>SubVector&	v.scatter(x, t),
<Type><System>SubMatrix&	M.scatter(x, t) and
<Type><System>SubTensor&	T.scatter(x, t)

respectively where vector arguments x and t are the same size. Function gather retrieves the elements $t_k = v_j$, $t_k = M_{i,j}$ or $t_k = T_{h,i,j}$ corresponding to index x_k then returns the const <Type><System>Vector t. Function scatter assigns the element $v_j = t_k$, $M_{i,j} = t_k$ or $T_{h,i,j} = t_k$ corresponding to index x_k then returns a reference to the subtensor.

Join two subtensors: All SubVector classes except offsetSubVector and all SubMatrix and SubTensor classes include constant member functions

const <Type><System>	functionality	conformance conditions
Vector	v.aside(w) $= [v \| w]$	
Matrix	M.aside(N) $= [M \| N]$	M.extent2() == N.extent2()
Tensor	T.aside(U) $= [T \| U]$	T.extent2() == U.extent2() T.extent3() == U.extent3()

which return a new tensor created by adjoining two subtensors.

All SubVector, SubMatrix and SubTensor classes except offsetSubVector include constant member functions

const <Type><System>	functionality	conformance conditions
Matrix	v.above(w) $= \begin{bmatrix} v \\ w \end{bmatrix}$	v.extent() == w.extent()
Matrix	v.above(M) $= \begin{bmatrix} v \\ M \end{bmatrix}$	M.extent1() == v.extent()
Matrix	M.above(N) $= \begin{bmatrix} M \\ N \end{bmatrix}$	M.extent1() == N.extent1()
Matrix	M.above(v) $= \begin{bmatrix} M \\ v \end{bmatrix}$	M.extent1() == v.extent()
Tensor	T.above(U) $= \begin{bmatrix} T \\ U \end{bmatrix}$	T.extent3() == U.extent3() T.extent1() == U.extent1()

which return a new tensor created by stacking two subtensors.

All `SubMatrix` and `SubTensor` classes include constant member functions

const <Type><System>	functionality	conformance conditions
Tensor	`M.afore(N)` $= [M]\ N]$	`M.extent1() == N.extent1()` `M.extent2() == N.extent2()`
Tensor	`M.afore(U)` $= [M]\ U]$	`M.extent1() == U.extent1()` `M.extent2() == U.extent2()`
Tensor	`T.afore(U)` $= [T]\ U]$	`T.extent1() == U.extent1()` `T.extent2() == U.extent2()`
Tensor	`T.afore(M)` $= [T]\ M]$	`T.extent1() == M.extent1()` `T.extent2() == M.extent2()`

which return a new tensor created by appending one subtensor to another.

Kronecker product: All `SubVector` classes except `offsetSubVector` and `boolSubVector` include constant member functions

return type	function	functionality
const <Type><System>Vector	`v.kron(w)`	$v \otimes w$,
const <Type><System>Matrix	`v.kron(M)`	$v \otimes M$ and
const <Type><System>Tensor	`v.kron(T)`	$v \otimes T$.

All `SubMatrix` classes except `boolSubMatrix` include constant member functions

return type	function	functionality
const <Type><System>Matrix	`M.kron(v)`	$M \otimes v$,
const <Type><System>Matrix	`M.kron(N)`	$M \otimes N$ and
const <Type><System>Tensor	`M.kron(T)`	$M \otimes T$.

All `SubTensor` classes except `boolSubTensor` include constant member functions

return type	function	functionality
const <Type><System>Tensor	`T.kron(v)`	$T \otimes v$,
const <Type><System>Tensor	`T.kron(M)`	$T \otimes M$ and
const <Type><System>Tensor	`T.kron(U)`	$T \otimes U$.

Apply univariate functions: All `SubVector`, `SubMatrix` and `SubTensor` classes except `offsetSubVector` include constant member functions

return type	function	functionality
const <Type><System>Vector	`v.apply(f)`	$f(v_j)$,
const <Type><System>Matrix	`M.apply(f)`	$f(M_{i,j})$ and
const <Type><System>Tensor	`T.apply(f)`	$f(T_{h,i,j})$

respectively which apply univariate functions

real	complex
<type> f(<type>)	<Type>Complex f(<Type>Complex) or
<type> f(const <type>&)	<Type>Complex f(const <Type>Complex&)

to the subtensor element by element.

Exponent, logarithm and square root: All real and complex floating-point SubVector, SubMatrix and SubTensor classes include functions

return type	exponent, logarithm and square root		
const <Type><System>Vector	exp(v)	log(v)	sqrt(v),
const <Type><System>Matrix	exp(M)	log(M)	sqrt(M) and
const <Type><System>Tensor	exp(T)	log(T)	sqrt(T)

respectively which are applied to the subtensor element by element.

Trigonometric and hyperbolic functions: All real and complex floating-point SubVector, SubMatrix and SubTensor classes include

return type	trigonometric functions		
const <Type><System>Vector	cos(v)	sin(v)	tan(v),
const <Type><System>Matrix	cos(M)	sin(M)	tan(M) and
const <Type><System>Tensor	cos(T)	sin(T)	tan(T)

respectively,

return type	inverse trigonometric functions		
const <Type><System>Vector	acos(v)	asin(v)	atan(v),
const <Type><System>Matrix	acos(M)	asin(M)	atan(M) and
const <Type><System>Tensor	acos(T)	asin(T)	atan(T)

respectively,

return type	hyperbolic functions		
const <Type><System>Vector	cosh(v)	sinh(v)	tanh(v),
const <Type><System>Matrix	cosh(M)	sinh(M)	tanh(M) and
const <Type><System>Tensor	cosh(T)	sinh(T)	tanh(T)

respectively and

return type	inverse hyperbolic functions		
const <Type><System>Vector	acosh(v)	asinh(v)	atanh(v),
const <Type><System>Matrix	acosh(M)	asinh(M)	atanh(M) and
const <Type><System>Tensor	acosh(T)	asinh(T)	atanh(T)

respectively which are applied to the subtensor element by element.

Sign and magnitude: All signed integer and real floating-point `SubVector`, `SubMatrix` and `SubTensor` classes include functions

return type	sign		magnitude	
`const <Type>Vector`	`sgn(v)`	$v_j/\lvert v_j\rvert$	`abs(v)`	$\lvert v_j\rvert$,
`const <Type>Matrix`	`sgn(M)`	$M_{i,j}/\lvert M_{i,j}\rvert$	`abs(M)`	$\lvert M_{i,j}\rvert$ and
`const <Type>Tensor`	`sgn(T)`	$T_{h,i,j}/\lvert T_{h,i,j}\rvert$	`abs(T)`	$\lvert T_{h,i,j}\rvert$

respectively which are applied to the subtensor element by element.

Floor and ceiling: All real floating-point `SubVector`, `SubMatrix` and `SubTensor` classes include functions

return type	floor		ceiling	
`const <Type>Vector`	`floor(v)`	$\lfloor v_j \rfloor$	`ceil(v)`	$\lceil v_j \rceil$,
`const <Type>Matrix`	`floor(M)`	$\lfloor M_{i,j} \rfloor$	`ceil(M)`	$\lceil M_{i,j} \rceil$ and
`const <Type>Tensor`	`floor(T)`	$\lfloor T_{h,i,j} \rfloor$	`ceil(T)`	$\lceil T_{h,i,j} \rceil$

respectively which are applied to the subtensor element by element.

Hypotenuse and arctangent: All real floating-point `SubVector`, `SubMatrix` and `SubTensor` classes include functions

return type	function	functionality
`const <Type>Vector`	`hypot(v, w)`	$\sqrt{v_j^2 + w_j^2}$,
`const <Type>Matrix`	`hypot(M, N)`	$\sqrt{M_{i,j}^2 + N_{i,j}^2}$ and
`const <Type>Tensor`	`hypot(T, U)`	$\sqrt{T_{h,i,j}^2 + U_{h,i,j}^2}$

respectively and

return type	function	functionality
`const <Type>Vector`	`atan2(w, v)`	$\tan^{-1}(w_j/v_j)$,
`const <Type>Matrix`	`atan2(N, M)`	$\tan^{-1}(N_{i,j}/M_{i,j})$ and
`const <Type>Tensor`	`atan2(U, T)`	$\tan^{-1}(U_{h,i,j}/T_{h,i,j})$

respectively which are applied to the subtensor element by element. Both arguments must be the same size.

Complex functions: All complex floating-point `SubVector`, `SubMatrix` and `SubTensor` classes include

return type	magnitude, argument and norm functions		
`const <Type>Vector`	`abs(v)`	`arg(v)`	`norm(v)`,
`const <Type>Matrix`	`abs(M)`	`arg(M)`	`norm(M)` and
`const <Type>Tensor`	`abs(T)`	`arg(T)`	`norm(T)`

respectively and

return type	complex conjugate and polar functions		
`const <Type>ComplexVector`	`conj(v)`	`iconj(v)`	`polar(v, w)`,
`const <Type>ComplexMatrix`	`conj(M)`	`iconj(M)`	`polar(M, N)` and
`const <Type>ComplexTensor`	`conj(T)`	`iconj(T)`	`polar(T, U)`

respectively which are applied to the subtensor element by element. Both arguments of the polar function must be the same size.

2.3.3 Operators

Subscript: If `v` is an n subvector, `M` is an $m \times n$ submatrix and `T` is an $l \times m \times n$ subtensor, `v[j]` returns column j as a subscalar, `M[i]` returns row i as a subvector and `T[h]` returns page h as a submatrix. This means that both `M[i][j]` and `T[h][i][j]` are subscalars. The elements referenced by the respective subscalars are at

	location
s	`s.offset()`
v_j	`v.offset() + j*v.stride()`
$M_{i,j}$	`M.offset() + j*M.stride1() + i*M.stride2()`
$T_{h,i,j}$	`T.offset() + j*T.stride1() + i*T.stride2() + h*T.stride3()`

where $0 \leq h < l$, $0 \leq i < m$ and $0 \leq j < n$. SVMT class libraries do not normally check whether any of the zero based indices, h, i or j are in range or not.

Unary: All integral `SubVector`, `SubMatrix` and `SubTensor` classes except the boolean subtensor classes and `offsetSubVector` include bitwise constant member functions for unary operators

return type	complement	
`const <Type>Vector`	`~v`	$\overline{v_j}$,
`const <Type>Matrix`	`~M`	$\overline{M_{i,j}}$ and
`const <Type>Tensor`	`~T`	$\overline{T_{h,i,j}}$

respectively.

Classes `boolSubVector`, `boolSubMatrix` and `boolSubTensor` include constant member functions for unary operators

return type	not	
`const boolVector`	`!v`	$\neg v_j$,
`const boolMatrix`	`!M`	$\neg M_{i,j}$ and
`const boolTensor`	`!T`	$\neg T_{h,i,j}$

respectively.

All `SubVector`, `SubMatrix` and `SubTensor` classes except `boolSubVector`, `boolSubMatrix`, `boolSubTensor` and `offsetSubVector` include constant member functions for unary operators

return type	minus	
const <Type><System>Vector	-v	$-v_j$,
const <Type><System>Matrix	-M	$-M_{i,j}$ and
const <Type><System>Tensor	-T	$-T_{h,i,j}$

respectively and

return type	plus	
const <Type><System>SubVector	+v	$+v_j$,
const <Type><System>SubMatrix	+M	$+M_{i,j}$ and
const <Type><System>SubTensor	+T	$+T_{h,i,j}$

respectively.

Cast: If s is a real subscalar, the cast (type conversion) operator (<type>)s returns a real scalar. If s is a complex subscalar, the cast (type conversion) operator (<Type>Complex)s returns a complex scalar.

Binary: All SubVector classes except offsetSubVector and boolSubVector include element by element arithmetic operators

result	scalar–vector		vector–vector		vector–scalar	
multiply	s*v	$s \cdot v_j$	v*w	$v_j \cdot w_j$	v*s	$v_j \cdot s$
divide	s/v	s/v_j	v/w	v_j/w_j	v/s	v_j/s
add	s + v	$s + v_j$	v + w	$v_j + w_j$	v + s	$v_j + s$
subtract	s - v	$s - v_j$	v - w	$v_j - w_j$	v - s	$v_j - s$

which return const <Type>ComplexVector if either operand is complex or const <Type>Vector if both operands are real.

All SubMatrix classes except boolSubMatrix include element by element arithmetic operators

result	scalar–matrix		matrix–matrix		matrix–scalar	
multiply	s*M	$s \cdot M_{i,j}$	M*N	$M_{i,j} \cdot N_{i,j}$	M*s	$M_{i,j} \cdot s$
divide	s/M	$s/M_{i,j}$	M/N	$M_{i,j}/N_{i,j}$	M/s	$M_{i,j}/s$
add	s + M	$s + M_{i,j}$	M + N	$M_{i,j} + N_{i,j}$	M + s	$M_{i,j} + s$
subtract	s - M	$s - M_{i,j}$	M - N	$M_{i,j} - N_{i,j}$	M - s	$M_{i,j} - s$

which return const <Type>ComplexMatrix if either operand is complex or const <Type>Matrix if both operands are real.

All SubTensor classes except boolSubTensor include element by element arithmetic operators

result	scalar–tensor		tensor–tensor		tensor–scalar	
multiply	s*T	$s \cdot T_{h,i,j}$	T*U	$T_{h,i,j} \cdot U_{h,i,j}$	T*s	$T_{h,i,j} \cdot s$
divide	s/T	$s/T_{h,i,j}$	T/U	$T_{h,i,j}/U_{h,i,j}$	T/s	$T_{h,i,j}/s$
add	s + T	$s + T_{h,i,j}$	T + U	$T_{h,i,j} + U_{h,i,j}$	T + s	$T_{h,i,j} + s$
subtract	s - T	$s - T_{h,i,j}$	T - U	$T_{h,i,j} - U_{h,i,j}$	T - s	$T_{h,i,j} - s$

which return const <Type>ComplexTensor if either operand is complex or const <Type>Tensor if both operands are real.

Of course, if x and y $\neq 0$ are integers, then

$$\mathtt{x/y} = \frac{x - x\%y}{y} \tag{2}$$

where

$$x\%y = \begin{cases} -(|x| \bmod |y|) & \text{if } x < 0 \text{ and} \\ +(|x| \bmod |y|) & \text{otherwise.} \end{cases} \tag{3}$$

All integral SubVector classes except boolSubVector and offsetSubVector include element by element arithmetic operators

result	scalar–vector		vector–vector		vector–scalar	
remainder	s%v	$s\%v_j$	v%w	$v_j\%w_j$	v%s	$v_j\%s$

which return const <Type>Vector.

All integral SubMatrix classes except boolSubMatrix include element by element arithmetic operators

result	scalar–matrix		matrix–matrix		matrix–scalar	
remainder	s%M	$s\%M_{i,j}$	M%N	$M_{i,j}\%N_{i,j}$	M%s	$M_{i,j}\%s$

which return const <Type>Matrix.

All integral SubTensor classes except boolSubTensor include element by element arithmetic operators

result	scalar–tensor		tensor–tensor		tensor–scalar	
remainder	s%T	$s\%T_{h,i,j}$	T%U	$T_{h,i,j}\%U_{h,i,j}$	T%s	$T_{h,i,j}\%s$

which return const <Type>Tensor.

All integral SubVector classes except offsetSubVector and boolSubVector and all integral SubMatrix and SubTensor classes except boolSubMatrix and boolSubTensor include bitwise shift operators

return type	shift left		shift right	
const <Type>Vector	v << n	$v_j << n$	v >> n	$v_j >> n$,
const <Type>Matrix	M << n	$M_{i,j} << n$	M >> n	$M_{i,j} >> n$ and
const <Type>Tensor	T << n	$T_{h,i,j} << n$	T >> n	$T_{h,i,j} >> n$

respectively which are applied to the subtensor element by element where argument $n \geq 0$ is an integer.

All SubVector, SubMatrix and SubTensor classes include input and output operators

ostream&		istream&	
cout << v	cout << v[j] << ' '	cin >> v	cin >> v[j],
cout << M	cout << M[i]	cin >> M	cin >> M[i] and
cout << T	cout << T[h]	cin >> T	cin >> T[h]

respectively. On output, each element of a row vector is followed by a space or a newline character so that there are no more than <Type><System>SubVector::columns() elements on each line.

All real SubVector classes except boolSubVector and offsetSubVector include comparison operators

scalar–vector		vector–vector		vector–scalar		
`s < v`	$s < v_j$	`v < w`	$v_j < w_j$	`v < s`	$v_j < s$	$\forall_j$,
`s <= v`	$s \leq v_j$	`v <= w`	$v_j \leq w_j$	`v <= s`	$v_j \leq s$	$\forall_j$,
`s > v`	$s > v_j$	`v > w`	$v_j > w_j$	`v > s`	$v_j > s$	$\forall_j$ and
`s >= v`	$s \geq v_j$	`v >= w`	$v_j \geq w_j$	`v >= s`	$v_j \geq s$	$\forall_j$

which return type bool.

All real SubMatrix classes except boolSubMatrix include comparison operators

scalar–matrix		matrix–matrix		matrix–scalar		
`s < M`	$s < M_{i,j}$	`M < N`	$M_{i,j} < N_{i,j}$	`M < s`	$M_{i,j} < s$	$\forall_{i,j}$,
`s <= M`	$s \leq M_{i,j}$	`M <= N`	$M_{i,j} \leq N_{i,j}$	`M <= s`	$M_{i,j} \leq s$	$\forall_{i,j}$,
`s > M`	$s > M_{i,j}$	`M > N`	$M_{i,j} > N_{i,j}$	`M > s`	$M_{i,j} > s$	$\forall_{i,j}$ and
`s >= M`	$s \geq M_{i,j}$	`M >= N`	$M_{i,j} \geq N_{i,j}$	`M >= s`	$M_{i,j} \geq s$	$\forall_{i,j}$

which return type bool.

All real SubTensor classes except boolSubTensor include comparison operators

scalar–tensor		tensor–tensor		tensor–scalar		
`s < T`	$s < T_{h,i,j}$	`T < U`	$T_{h,i,j} < U_{h,i,j}$	`T < s`	$T_{h,i,j} < s$	$\forall_{h,i,j}$,
`s <= T`	$s \leq T_{h,i,j}$	`T <= U`	$T_{h,i,j} \leq U_{h,i,j}$	`T <= s`	$T_{h,i,j} \leq s$	$\forall_{h,i,j}$,
`s > T`	$s > T_{h,i,j}$	`T > U`	$T_{h,i,j} > U_{h,i,j}$	`T > s`	$T_{h,i,j} > s$	$\forall_{h,i,j}$ and
`s >= T`	$s \geq T_{h,i,j}$	`T >= U`	$T_{h,i,j} \geq U_{h,i,j}$	`T >= s`	$T_{h,i,j} \geq s$	$\forall_{h,i,j}$

which return type bool.

All SubVector classes include comparison operators

scalar–vector		vector–vector		vector–scalar		
`s == v`	$s = v_j$	`v == w`	$v_j = w_j$	`v == s`	$v_j = s$	$\forall_j$ and
`s != v`	$s \neq v_j$	`v != w`	$v_j \neq w_j$	`v != s`	$v_j \neq s$	$\forall_j$

which return type bool.

All SubMatrix classes include comparison operators

scalar–matrix		matrix–matrix		matrix–scalar		
`s == M`	$s = M_{i,j}$	`M == N`	$M_{i,j} = N_{i,j}$	`M == s`	$M_{i,j} = s$	$\forall_{i,j}$ and
`s != M`	$s \neq M_{i,j}$	`M != N`	$M_{i,j} \neq N_{i,j}$	`M != s`	$M_{i,j} \neq s$	$\forall_{i,j}$

which return type bool.

All SubTensor classes include comparison operators

scalar–tensor		tensor–tensor		tensor–scalar		
`s == T`	$s = T_{h,i,j}$	`T == U`	$T_{h,i,j} = U_{h,i,j}$	`T == s`	$T_{h,i,j} = s$	$\forall_{h,i,j}$ and
`s != T`	$s \neq T_{h,i,j}$	`T != U`	$T_{h,i,j} \neq U_{h,i,j}$	`T != s`	$T_{h,i,j} \neq s$	$\forall_{h,i,j}$

which return type `bool`.

All integral `SubVector` classes except `boolSubVector` and `offsetSubVector` include element by element bitwise operators

operation	scalar–vector		vector–vector		vector–scalar	
AND	`s&v`	$s \& v_j$	`v&w`	$v_j \& w_j$	`v&s`	$v_j \& s$
XOR	`s^v`	$s \hat{} v_j$	`v^w`	$v_j \hat{} w_j$	`v^s`	$v_j \hat{} s$
OR	`s|v`	$s \vert v_j$	`v|w`	$v_j \vert w_j$	`v|s`	$v_j \vert s$

which return `const <Type>Vector`.

All integral `SubMatrix` classes except `boolSubMatrix` include element by element bitwise operators

operation	scalar–matrix		matrix–matrix		matrix–scalar	
AND	`s&M`	$s \& M_{i,j}$	`M&N`	$M_{i,j} \& N_{i,j}$	`M&s`	$M_{i,j} \& s$
XOR	`s^M`	$s \hat{} M_{i,j}$	`M^N`	$M_{i,j} \hat{} N_{i,j}$	`M^s`	$M_{i,j} \hat{} s$
OR	`s|M`	$s \vert M_{i,j}$	`M|N`	$M_{i,j} \vert N_{i,j}$	`M|s`	$M_{i,j} \vert s$

which return `const <Type>Matrix`.

All integral `SubTensor` classes except `boolSubTensor` include element by element bitwise operators

operation	scalar–tensor		tensor–tensor		tensor–scalar	
AND	`s&T`	$s \& T_{h,i,j}$	`T&U`	$T_{h,i,j} \& U_{h,i,j}$	`T&s`	$T_{h,i,j} \& s$
XOR	`s^T`	$s \hat{} T_{h,i,j}$	`T^U`	$T_{h,i,j} \hat{} U_{h,i,j}$	`T^s`	$T_{h,i,j} \hat{} s$
OR	`s|T`	$s \vert T_{h,i,j}$	`T|U`	$T_{h,i,j} \vert U_{h,i,j}$	`T|s`	$T_{h,i,j} \vert s$

which return `const <Type>Tensor`. Both operands of all vector–vector, matrix-matrix and tensor-tensor binary operations must be the same size.

Assignment: All `SubVector` classes include simple assignment operators

return type	vector–vector		vector–scalar	
`<Type><System>SubVector&`	`v = w`	$v_j \leftarrow w_j$	`v = s`	$v_j \leftarrow s$

which return a reference to the left hand side.

All `SubMatrix` and `SubTensor` classes include simple assignment operators

return type	matrix–matrix		matrix–scalar	
`<Type><System>SubMatrix&`	`M = N`	$M_{i,j} \leftarrow N_{i,j}$	`M = s`	$M_{i,j} \leftarrow s$ and
return type	**tensor–tensor**		**tensor–scalar**	
`<Type><System>SubTensor&`	`T = U`	$T_{h,i,j} \leftarrow U_{h,i,j}$	`T = s`	$T_{h,i,j} \leftarrow s$

respectively which return a reference to the left hand side.

All SubVector classes except offsetSubVector and boolSubVector include assignment operators

operation	vector–vector		vector–scalar	
multiply	v *= w	$v_j \leftarrow v_j \cdot w_j$	v *= s	$v_j \leftarrow v_j \cdot s$,
divide	v /= w	$v_j \leftarrow v_j / w_j$	v /= s	$v_j \leftarrow v_j / s$,
add	v += w	$v_j \leftarrow v_j + w_j$	v += s	$v_j \leftarrow v_j + s$ and
subtract	v -= w	$v_j \leftarrow v_j - w_j$	v -= s	$v_j \leftarrow v_j - s$

return a reference of type <Type><System>SubVector& to the left hand side.

All SubMatrix classes except boolSubMatrix include assignment operators

operation	matrix–matrix		matrix–scalar	
multiply	M *= N	$M_{i,j} \leftarrow M_{i,j} \cdot N_{i,j}$	M *= s	$M_{i,j} \leftarrow M_{i,j} \cdot s$,
divide	M /= N	$M_{i,j} \leftarrow M_{i,j} / N_{i,j}$	M /= s	$M_{i,j} \leftarrow M_{i,j} / s$,
add	M += N	$M_{i,j} \leftarrow M_{i,j} + N_{i,j}$	M += s	$M_{i,j} \leftarrow M_{i,j} + s$ and
subtract	M -= N	$M_{i,j} \leftarrow M_{i,j} - N_{i,j}$	M -= s	$M_{i,j} \leftarrow M_{i,j} - s$

return a reference of type <Type><System>SubMatrix& to the left hand side.

All SubTensor classes except boolSubTensor include assignment operators

operation	tensor–tensor		tensor–scalar	
multiply	T *= U	$T_{h,i,j} \leftarrow T_{h,i,j} \cdot U_{h,i,j}$	T *= s	$T_{h,i,j} \leftarrow T_{h,i,j} \cdot s$,
divide	T /= U	$T_{h,i,j} \leftarrow T_{h,i,j} / U_{h,i,j}$	T /= s	$T_{h,i,j} \leftarrow T_{h,i,j} / s$,
add	T += U	$T_{h,i,j} \leftarrow T_{h,i,j} + U_{h,i,j}$	T += s	$T_{h,i,j} \leftarrow T_{h,i,j} + s$ and
subtract	T -= U	$T_{h,i,j} \leftarrow T_{h,i,j} - U_{h,i,j}$	T -= s	$T_{h,i,j} \leftarrow T_{h,i,j} - s$

return a reference of type <Type><System>SubTensor& to the left hand side.

All integral <Type>SubVector, <Type>SubMatrix and <Type>SubTensor classes include modulo and assign operators

return type	vector–vector		vector–scalar	
<Type>SubVector&	v %= w	$v_j \leftarrow v_j \% w_j$	v %= s	$v_j \leftarrow v_j \% s$,
return type	matrix–matrix		matrix–scalar	
<Type>SubMatrix&	M %= N	$M_{i,j} \leftarrow M_{i,j} \% N_{i,j}$	M %= s	$M_{i,j} \leftarrow M_{i,j} \% s$ and
return type	tensor–tensor		tensor–scalar	
<Type>SubTensor&	T %= U	$T_{h,i,j} \leftarrow T_{h,i,j} \% U_{h,i,j}$	T %= s	$T_{h,i,j} \leftarrow T_{h,i,j} \% s$

respectively which return a reference to the left hand side.

Both operands of all vector–vector, matrix-matrix and tensor-tensor assignment operations must be the same size.

2.4 SubArray0, SubArray1, SubArray2 and SubArray3 classes

The SubArray0, SubArray1, SubArray2 and SubArray3 classes inherit everything from SubScalar, SubVector, SubMatrix and SubTensor classes respectively except the constructors and resize member functions.

2.4.1 Constructors

Explicit Constructors

`<Type><System>SubArray0 s(p, o)`,
`<Type><System>SubArray1 v(p, o, n1, s1)`,
`<Type><System>SubArray2 M(p, o, n2, s2, n1, s1)` and
`<Type><System>SubArray3 T(p, o, n3, s3, n2, s2, n1, s1)`

permit the application programmer to reference the elements of an ordinary one dimensional array with a SubScalar, SubVector, SubMatrix or SubTensor respectively by passing a pointer `p` of type `<type>*` to the beginning of the one dimensional array, an offset `o` of type `Offset` from the beginning of the one dimensional array to the first element of the subtensor and a stride `s1`, `s2` or `s3` of type `Stride` and an extent `n1`, `n2` or `n3` of type `Extent` for each dimension. The constructors can not check whether the one dimensional array actually contains the subtensor or not.

Copy Constructors

`<Type><System>SubArray0 t(s)`,
`<Type><System>SubArray1 w(v)`,
`<Type><System>SubArray2 N(M)` and
`<Type><System>SubArray3 U(T)`

simply copy the attributes of an existing subarray not the underlying one dimensional array.

Default Constructors

`<Type><System>SubArray1 v`,
`<Type><System>SubArray2 M` and
`<Type><System>SubArray3 T`

create an empty subvector, submatrix and subtensor respectively.

The SVMT class library developer may derive `SubArray0`, `SubArray1`, `SubArray2` and `SubArray3` classes from `SubScalar`, `SubVector`, `SubMatrix` and `SubTensor` classes respectively or implement them as unrelated classes with equivalent functionality.

2.4.2 Functions

All `SubArray1`, `SubArray2` and `SubArray3` classes include member functions

return type	explicit resize
`<Type><System>SubArray1&`	`v.resize(p, o, n1, s1)`,
`<Type><System>SubArray2&`	`M.resize(p, o, n2, s2, n1, s1)` and
`<Type><System>SubArray3&`	`T.resize(p, o, n3, s3, n2, s2, n1, s1)`

respectively which correspond to the respective explicit constructors.

All `SubArray1`, `SubArray2` and `SubArray3` classes include member functions

return type	copy resize
`<Type><System>SubArray1&`	`w.resize(v)`,
`<Type><System>SubArray2&`	`N.resize(M)` and
`<Type><System>SubArray3&`	`U.resize(T)`

respectively which correspond to the respective copy constructors.

All `SubArray1`, `SubArray2` and `SubArray3` classes include member functions

return type	default resize
`<Type><System>SubArray1&`	`v.resize()`,
`<Type><System>SubArray2&`	`M.resize()` and
`<Type><System>SubArray3&`	`T.resize()`

respectively which correspond to the respective default constructors.

2.5 Vector, Matrix and Tensor classes

The Vector, Matrix and Tensor classes are derived from the SubVector, SubMatrix and SubTensor classes respectively. Their constructors allocate a one dimensional array for them automatically when they are created and their destructors deallocate it automatically for them when they are destroyed. Otherwise, they inherit everything from their respective base classes except constructors and resize and free member functions.

2.5.1 Constructors

Explicit Constructors

`<Type><System>Vector v(n)`,
`<Type><System>Matrix M(m, n)` and
`<Type><System>Tensor T(l, m, n)`

create an uninitialized n vector, $m \times n$ matrix and $l \times m \times n$ tensor respectively. The second and third arguments default to a value of +1 if omitted.

Explicit Constructors

<Type><System>Vector v(n, s),
<Type><System>Matrix M(m, n, s) and
<Type><System>Tensor T(l, m, n, s)

create an n vector, $m \times n$ matrix and $l \times m \times n$ tensor respectively and initialize each element to scalar value s.

Explicit Constructors

<Type>Vector v(n, s, t),
<Type>Matrix M(m, n, s, t) and
<Type>Tensor T(l, m, n, s, t)

create a real n vector, $m \times n$ matrix and $l \times m \times n$ tensor respectively then initialize each row such that $v_j = s + j \cdot t$, $M_{i,j} = s + j \cdot t$ and $T_{h,i,j} = s + j \cdot t$.

Copy constructors

<Type><System>Vector w = v,
<Type><System>Matrix N = M and
<Type><System>Tensor U = T

create a new vector w, a new matrix N and a new tensor U the same size as vector v, matrix M and tensor T respectively and copy every element of vector v to vector w, matrix M to matrix N and tensor T to tensor U.

Default constructors

<Type><System>Vector v,
<Type><System>Matrix M and
<Type><System>Tensor T

create an empty vector, matrix and tensor respectively.

Explicit Constructors

<Type>ComplexVector w = v,
<Type>ComplexMatrix N = M and
<Type>ComplexTensor U = T

compose a new complex vector w, matrix N and tensor U from a real subvector v, a real submatrix M and a real subtensor T respectively. The imaginary parts are set to zero.

Explicit Constructors

`<Type>ComplexVector w(v, u)`,
`<Type>ComplexMatrix N(M, L)` and
`<Type>ComplexTensor U(T, S)`

compose a new complex vector w, matrix N and tensor U from a pair of real subvectors v and u, a pair of real submatrices M and L and a pair of real subtensors T and S respectively which represent the real and imaginary parts.

2.5.2 Functions

All `Vector`, `Matrix` and `Tensor` classes include member functions

return type	`<Type><System>` `Vector&`	`<Type><System>` `Matrix&`	`<Type><System>` `Tensor&`
explicit	`v.resize(n)`	`M.resize(m, n)`	`T.resize(l, m, n)`,
explicit	`v.resize(n, s)`	`M.resize(m, n, s)`	`T.resize(l, m, n, s)`,
explicit	`v.resize(n, s, t)`	`M.resize(m, n, s, t)`	`T.resize(l, m, n, s, t)`,
copy	`w.resize(v)`	`N.resize(M)`	`U.resize(T)` and
default	`v.resize()`	`M.resize()`	`T.resize()`

and all complex `Vector`, `Matrix` and `Tensor` classes include member functions

return type	`<Type>Complex` `Vector&`	`<Type>Complex` `Matrix&`	`<Type>Complex` `Tensor&`
explicit	`w.resize(v)`	`N.resize(M)`	`U.resize(T)` and
explicit	`w.resize(v, u)`	`N.resize(M, L)`	`U.resize(T, S)`

corresponding to each of the constructors.

The resize function may deallocate the original one dimensional array before it allocates a new one dimensional array and all the data may be lost. Any subtensor of a tensor created before the resize function is applied to the tensor references a one dimensional array which may no longer exist after the resize function returns.

The application programmer should avoid using the resize function but it is essential for tasks like creating an array of tensors which have different sizes. The default constructor is used to create the array of tensors then the resize function is applied to each element in order to size it appropriately.

2.6 SubSquare and Square (Matrix) classes

2.6.1 SubSquare classes

The SubSquare classes inherit almost all of the functions and operations from the respective SubMatrix classes but `<Type><System>SubSquare` functions and operations return `boolSquare`, `<Type><System>Square`, `<Type><System>SubSquare` or `<Type><System>SubSquare&` where `<Type><System>SubMatrix` functions and operations return `boolMatrix`, `<Type><System>Matrix`, `<Type><System>SubMatrix` or `<Type><System>SubMatrix&` respectively whenever the result must be a square matrix. The explicit SubSquare constructor

`<Type><System>SubSquare M(h, o, m, s2, s1)`,

static member function allocate

return type	allocate memory
`<Type><System>Handle`	`allocate(m)`

and member function resize

return type	explicit resize
`<Type><System>SubSquare&`	`M.resize(h, o, m, s2, s1)`

were redefined so that all SubSquare objects are created with `extent1() == m == extent2()`.

All SubMatrix classes include member functions

return type	function
`<Type><System>SubSquare`	`M.subsquare(i, m, s2, j, s1)`

where stride $s_1 = 1$ if omitted and offset $j = 0$ if omitted which return a reference to a square submatrix of a submatrix.

The SubSquare classes include functions which apply only to square matrices.

The transpose

return type	transpose
`<Type><System>SubSquare&`	`M.transpose()`

member function actually transposes the elements of the original square submatrix `M` in place and returns a reference to `M`.

Cholesky decomposition: All real and complex floating-point `SubSquare` classes include member function

return type	function	solves
`const offsetVector`	`p = M.lld()`	$PMP^T = (LD)(LD)^T$

which decomposes symmetric matrix M in-place ignoring the superdiagonal elements. It returns the permutation vector `p` required to obtain the permutation PMP^T of the rows and columns of symmetric matrix M. The superdiagonal elements of M are left unchanged but matrix LD is packed into the remaining elements of M. If symmetric matrix M is real, the result is undefined unless M is positive definite.

The inverse

return type	inverse
const <Type><System>Square	M.i(r)

member function returns a square matrix which is the inverse of the original floating-point square submatrix M without disturbing it but member function

return type	inverse
<Type><System>SubSquare&	M.invert(r)

inverts the original floating-point square submatrix M in place and returns a reference to M. An optional argument r of type <type> defaults to zero if omitted. The application programmer must provide the actual definition for both of these member functions.

2.6.2 Square classes

The Square classes inherit all of the functions and operations from the respective SubSquare classes except constructors and resize member functions. The explicit Square constructors

```
<Type><System>Square M(m),
<Type><System>Square M(m, s) and
<Type><System>Square M(m, s, t)
```

create an uninitialized $m{\times}m$ square matrix, an $m{\times}m$ square matrix initialized with $M_{i,j} = s$ and an $m \times m$ square matrix initialized with $M_{i,j} = s + j \cdot t$ respectively and member functions

return type	explicit resize
<Type><System>Square&	M.resize(m),
<Type><System>Square&	M.resize(m, s) and
<Type><System>Square&	M.resize(m, s, t)

are the respective reconstructors.

Neither SubSquare or Square objects can be constructed or reconstructed from SubMatrix or Matrix objects.

3 Convention

There are two basic differences between C and Fortran style arrays. First, Fortran subscripts begin with 1 but C subscripts begin with 0. Second, C subscripts appear in reverse order from Fortran subscripts so C style arrays appear to be the transpose of Fortran style arrays. Consequently, it is natural for C programmers to think of one dimensional arrays as row vectors instead of column vectors and to think of two dimensional arrays as collections of row vectors instead of collections of column vectors. This point of view has certain practical and conceptual advantages. For example, the elements of a two dimensional C style array are stored in memory in the same order that they appear in the I/O stream.

It is easy to convert a Fortran style column vector oriented expression into a C style row vector oriented expression. Just transpose the expression then replace each vector and matrix with its transpose so that the Fortran style matrix-vector dot product $b = A^T x$ becomes the C style vector-matrix dot product $b = xA^T$ for example. The conversion has no effect on element-by-element functions and operations.

In order to accommodate Fortran programmers with a minimum of inconvenience, `operator ()` was overloaded and alternate forms of some member functions were provided to support Fortran style subscripts. Fortran programmers are still obliged to reverse the extent arguments in explicit constructors such as

$m \times n$ matrix M	<Type><System>Matrix M(n, m)
$m \times n \times p$ tensor T	<Type><System>Tensor T(p, n, m)

or to define pseudo constructors like this:

```
inline
const
<Type><System>Matrix <Type><System>Matrix_(Extent m, Extent n) {
  return <Type><System>Matrix(n, m); }
```

which can be used like this:

```
<Type><System>Matrix M = <Type><System>Matrix_(m, n);
```

From the Fortran programmers' point of view, the function `p = M.lud()` appears to solve $MP^T = (LD)U$ so they might use triangular system solvers `x = v.up_(M, p)` and `w = x.ld_(M)` to solve $v = M^T w = (UP)^T (LD)^T w = (UP)^T x$.

The following tables cross reference the original and alternate forms of member operators and functions where `i`, `j`, `k` and `l` $\in \{1, 2, \ldots\}$ are indices, `n1`, `n2` and `n3` $\in \{0, 1, 2, \ldots\}$ are extents, `s1`, `s2` and `s3` $\in \{\ldots, -2, -1, +0, +1, +2, \ldots\}$ are strides, `h` is a handle, `o` is an offset, `s` is a scalar, `p` is a permutation vector object, `v` and `w` are vector objects, `M` and `N` are matrix objects and `T` and `U` are tensor objects.

v(i)	v[i-1]
M(j)	M[j-1]
M(i, j)	M[j-1][i-1]
T(k)	T[k-1]
T(j, k)	T[k-1][j-1]
T(i, j, k)	T[k-1][j-1][i-1]

v.contains_(i, n1, s1)	v.contains(i-1, n1, s1)
M.contains_(j, n2, s2)	M.contains(j-1, n2, s2)
M.contains_(i, n1, s1, j, n2, s2)	M.contains(j-1, n2, s2, i-1, n1, s1)
T.contains_(k, n3, s3)	T.contains(k-1, n3, s3)
T.contains_(j, n2, s2, k, n3, s3)	T.contains(k-1, n3, s3, j-1, n2, s2)
T.contains_(i, n1, s1, j, n2, s2, k, n3, s3)	T.contains(k-1, n3, s3, j-1, n2, s2, i-1, n1, s1)

v.sub_(i, n1, s1)	v.sub(i-1, n1, s1)
M.sub_(j, n2, s2)	M.sub(j-1, n2, s2)
M.sub_(i, n1, s1, j, n2, s2)	M.sub(j-1, n2, s2, i-1, n1, s1)
T.sub_(k, n3, s3)	T.sub(k-1, n3, s3)
T.sub_(j, n2, s2, k, n3, s3)	T.sub(k-1, n3, s3, j-1, n2, s2)
T.sub_(i, n1, s1, j, n2, s2, k, n3, s3)	T.sub(k-1, n3, s3, j-1, n2, s2, i-1, n1, s1)

M.subsquare_(j, n2, s2)	M.subsquare(j-1, n2, s2)
M.subsquare_(i, s1, j, n2, s2)	M.subsquare(j-1, n2, s2, i-1, s1)

v.swap_(i, l)	v.swap(i-1, l-1)
M.swap_(j, l)	M.swap(j-1, l-1)
T.swap_(k, l)	T.swap(k-1, l-1)

M.resize_(h, o, n1, s1, n2, s2)	M.resize(h, o, n2, s2, n1, s1)
T.resize_(h, o, n1, s1, n2, s2, n3, s3)	T.resize(h, o, n3, s3, n2, s2, n1, s1)

M.allocate_(n1, n2)	M.allocate(n2, n1)
T.allocate_(n1, n2, n3)	T.allocate(n3, n2, n1)

M.resize_(n1, n2)	M.resize(n2, n1)
M.resize_(n1, n2, s)	M.resize(n2, n1, s)
M.resize_(n1, n2, s, t)	M.resize(n2, n1, s, t)
T.resize_(n2, n3)	T.resize(n3, n2)
T.resize_(n1, n2, n3)	T.resize(n3, n2, n1)
T.resize_(n1, n2, n3, s)	T.resize(n3, n2, n1, s)
T.resize_(n1, n2, n3, s, t)	T.resize(n3, n2, n1, s, t)

v.even_()	v.odd()
M.even_()	M.odd()
T.even_()	T.odd()

v.odd_()	v.even()
M.odd_()	M.even()
T.odd_()	T.even()

v.min_()	1 + v.min()
v.max_()	1 + v.max()

v.dot_(w)	w.dot(v)
v.dot_(M)	M.dot(v)
M.dot_(v)	v.dot(M)
M.dot_(N)	N.dot(M)

v.pl_(p, M)	v.up(M, p)
v.up_(M, p)	v.pl(p, M)

v.l_(M)	v.u(M)
v.u_(M)	v.l(M)

v.pld_(p, M)	v.dup(M, p)
v.dup_(M, p)	v.pld(p, M)

v.ld_(M)	v.du(M)
v.du_(M)	v.ld(M)

v.above_(w)	v.aside(w)
v.aside_(w)	v.above(w)
v.aside_(M)	v.above(M)
M.aside_(v)	M.above(v)
M.aside_(N)	M.above(N)
M.above_(N)	M.aside(N)
T.above_(U)	T.aside(U)
T.aside_(U)	T.above(U)

4 Error Detection

Application programmers try to detect and eliminate all fatal programming errors before the application is placed into service because, even if they can be detected efficiently, there probably isn't much that can be done about them after the application is placed into service except to log them on some sort of "black box" recorder and restart the application.

Most of the math errors inherited from built-in types cannot be detected until run-time. Other programming errors cannot be detected until run-time because vector, matrix and tensor size information is not generally available at compile-time and the one dimensional arrays which they reference must be allocated and deallocated dynamically. These errors include containment, range, conformance, reference and memory errors.

5 Optimization

References

[1] Gene H. Golub and Charles F. Van Loan. *Matrix Computations Second Edition.* The Johns Hopkins University Press, 1989.

[2] William H. Press, Saul A. Teukolsky, William T. Vettering, and Brian P. Flannery. *Numerical Recipes in C The Art of Scientific Computing Second Edition.* Cambridge University Press, 1992.

[3] Bjarne Stroustrup. *The C++ Programming Language Third Edition.* Addison-Wesley Publishing Company, September 1997.

Index